# 配置自动编译vscode

|  |
| --- |
| 1、npm install -g typescript或者 yarn global add typescript  2. tsc --init 生成tsconfig.json 改 "outDir": "./js",  3、终端 -> 运行任务或ctlr+shift+b 监视tsconfig.json    4、 <script src="js/test.js"></script><!--自行添加js引入-->  -----------------------------------------------------------------------  tsc filename.ts 生成js  tsc filename.ts -d 生成声明文件filename.d.ts 代码提示功能 |

# 基础类型

|  |
| --- |
| typescript中为了使编写的代码更规范，更有利于维护，增加了类型校验，  在typescript中主要给我们提供了以下数据类型  typescript中为了使编写的代码更规范，更有利于维护，增加了类型校验  写ts代码必须指定类型 |

## 布尔类型(boolean)

|  |
| --- |
| *let* flag:*boolean* = true  flag = false  //正确 |

## 数字类型(number)

|  |
| --- |
| *let* num:*number* = 123  num = 456 |

## 字符串类型(string)

|  |
| --- |
| *let* str:*string* = 'hello'  str = 'world' |

## 数组类型(array)

|  |
| --- |
| // 第一种  *let* arr:*number*[] = [11, 22, 33]  // 第二种  *let* arr:*Array*<*number*> = [11, 22, 33]  // 第三种  *let* arr:*any*[] = ['11', 22, true] |

## 元组类型(tuple)

|  |
| --- |
| // 元组类型(tuple)  属于数组的一种  *let* arr:[*number*, *string*]=[11,'22', 33] |

## 枚举类型(enum)

|  |
| --- |
| // 1.数字枚举  // 枚举下标默认从0开始，赋值，则后续值递增  *enum* Color {      blue,      red = 10,      orange  }  *let* b:Color = Color.blue  *let* r:Color = Color.red  *let* o:Color = Color.orange  console.log(b) // 0  console.log(r) // 10  console.log(o) // 11  // 2.字符串枚举  // 在一个字符串枚举里，每个成员都必须用字符串字面量，或另外一个字符串枚举成员进行初始化  *enum* Direction {      NORTH = 'NORTH',      SOUTH = 'SOUTH',      EAST = 'EAST',      WEST = 'WEST'  }  // 3.异构枚举, 异构枚举的成员值是数字和字符串的混合  *enum* Enum {      A,      B,      C = "C",      D = "D",      E = 8,      F,  }  // 数字枚举相对字符串枚举多了 “反向映射”  console.log(Enum.A) //输出：0  console.log(Enum[0]) // 输出：A |

## 任意类型(any)

|  |
| --- |
| *let* num:*any* = 10  num = 'str'  num = true  //任意类型的用处  *let* oBox:*any* = document.getElementById('box')  oBox.style.color = 'red' |

## null和undefined

|  |
| --- |
| 默认情况下 null 和 undefined 是所有类型的子类型。  就是说你可以把 null 和 undefined 赋值给 number 类型的变量。  然而，如果你指定了--strictNullChecks 标记，null 和 undefined 只能赋值给 void 和它们各自的类型  // 定义没赋值就是undefined  // let num:number  console.log(num)  // undefined 报错  // let num:undefined  console.log(num) // undefined 正确  //可能是 number null undefined类型  *let* num:*number* | *null* | *undefined*  num = 10  console.log(num) // 10 |

## void类型

|  |
| --- |
| void表示没有任何类型，一般用于方法没有返回任何类型  *function* func():*void*{      console.log('hello')  }  //错误写法  *function* func(): *undefined*{      console.log('hello')  } |

## never类型

|  |
| --- |
| // never类型:是其他类型(包括null和undefined)的子类型，代表从不会出现的值  //这意味着声明never的变量只能被never类型所赋值  *let* foo:*never*  // foo =123 //错误写法  foo = (() *=>* {      throw new *Error*('错误')  })() |

# 函数

## 函数的定义

|  |
| --- |
| // es5函数声明法  *function* func(*name*) {      return 'func'  }  // 匿名函数  *var* func = *function*(*name*) {      return 'func'  }  //没有返回值的方法  *function* func() {      console.log('func')  }  ===========================================================  // ts中定义函数的方法  //函数声明法  *function* func(*name*:*string*):*string* {      return 'func'  }  // 匿名函数  *let* func = *function*(*name*:*string*):*string* {      return 'func'  }  //没有返回值的方法  *function* func():*void* {      console.log('func')  } |

## 可选参数

|  |
| --- |
| // es5方法的实参和形参可以不一样，  // ts中必须一样，不一样就需要配置可选参数  // 注意:可选参数必须配置到参数的最后面  *function* func(*name*:*string*, *age*?:*number*):*string* {      return *age* ? `${*name*} ${*age*}` : *name*  }  func('zhangsan') // zhangsan  func('zhangsan', 20) // zhangsan 20 |

## 默认参数

|  |
| --- |
| // es5不能设置默认参数  // es6和ts中都可以设置默认参数  *function* func(*name*:*string*, *age*:*number*=20):*string* {     return *age* ? `${*name*} ${*age*}` : *name*  }  func('zhangsan', 20) // zhangsan 20 |

## 剩余参数 也叫可变参数

|  |
| --- |
| // 三点运算符 接受新参传过来的值  *function* func(*a*:*number*, *b*:*number*, ...*result*:*number*[]):*number* {  *let* sum = *a* + *b*      for(*let* i=0; i<*result*.length; i++) {          sum += *result*[i]      }      return sum  }  console.log(func(1,2,3,4,5,6)) // 21 |

## 函数重载

|  |
| --- |
| //es5中出现同名方法，下面会替换上面的方法  *function* func(*a*) {...}  *function* func(*a*, *b*) {...}  // ts重载: 函数名相同，参数类型或参数个数不一样  // 方法是为同一个函数提供多个函数类型定义来进行函数重载  *function* func(*name*:*string*):*string*  function func(*name*:*string*, *age*:*number*):*string*  function func(*name*:*any*, *age*?:*any*):*any* {      return *age* ? `${*name*} ${*age*}` : *name*  }  console.log(func('zhangsan'))  // 正确  console.log(func('zhangsan', 20)) // 正确  console.log(func(123))  // 错误 |

## 箭头函数

|  |
| --- |
| // es5  setTimeout(*function*() {      console.log(this) // 指向window  }, 1000)  *let* obj = {   name: '张三',      func: *function*() {          setTimeout(*function*() {              console.log(this) // 指向window          }, 1000)      }  }  console.log(obj.func())  ===============================================================  // es6 注意：箭头函数里面的this指向上下文  setTimeout(() *=>* {      console.log(this) // 指向window  }, 1000)  *let* obj = {      name: '张三',      func: *function*() {          setTimeout(() *=>* {              console.log(this) // 指向obj          }, 1000)      }  }  console.log(obj.func()) |

# ES5类、对象、继承

## 构造函数和原型链增加方法

|  |
| --- |
| *function* Person(*name*){    // 构造函数Person里的属性、实例方法不会被多个实例共享    this.name = *name*  // 实例方法    this.run = *function*() {        console.log(this.name + '...run')    }  }  //原型链上面的属性、方法会被多个实例共享  Person.prototype.sex = '男'  Person.prototype.work = *function*() {    console.log(this.name + '...work')  }  *let* p1 = new Person('张三')  *let* p2 = new Person('李四')  console.log(p1)  console.log(p2)  console.log(p1.name === p2.name) // false  console.log(p1.run === p2.run) // false  console.log(p1.sex === p2.sex) // true  console.log(p1.work === p2.work) // true |

## 类的静态方法

|  |
| --- |
| *function* Person(){    // 属性    this.name = '张三'    // 实例方法    this.run = *function*() {      console.log(this.name + '...run')    }  }  Person.getInfo = *function*() {    console.log('静态方法')  }  Person.prototype.sex = '男'  Person.prototype.work = *function*(){    console.log(this.name + '...work')  }  *let* p = new Person()  // 调用实例方法  p.work()  //调用静态方法  Person.getInfo() |

## 对象冒充实现继承

|  |
| --- |
| 问题：实例化子类不能继承原型链上的属性和方法  *function* Person() {    this.name = '张三'    this.run = *function* () {      console.log(this.name + '...run')    }  }  Person.prototype.sex = '男'  Person.prototype.work = *function* () {    console.log(this.name + '...work')  }  // 对象冒充实现继承  *function* Student() {    Person.call(this)  }  *let* student = new Student()  // 对象冒充实现继承,可以继承构造函数里面的属性和实例方法  student.run() // 张三...run  // 但实例化子类不能继承原型链上的属性和方法  student.work() // 报错 work is not a function |

## 原型链继承

|  |
| --- |
| *function* Person() {    this.name = '张三'    this.run = *function*() {        console.log(this.name + '...run')    }  }  Person.prototype.sex = '男'  Person.prototype.work = *function*(){      console.log(this.name + '...work')  }  // 原型链继承: 可以继承构造函数里的属性和实例方法 也可以继承原型链上的属性和方法  *function* Student() { }  Student.prototype = new Person()  *let* student = new Student()  student.run()  student.work() |

## 原型链继承问题

|  |
| --- |
| 问题：子类的时候没法给父类传参  *function* Person(*name*, *age*) {    this.name = *name*    this.age = *age*    this.run = *function*() {        console.log(this.name + '...run')    }  }  Person.prototype.sex = '男'  Person.prototype.work = *function*(){    console.log(this.name + '...work')  }  *function* Student(*name*,*age*) { }  // 原型链继承  Student.prototype = new Person()  // 实例化子类的时候没法给父类传参  *let* student = new Student('赵四', 20)  student.run() // undefined...run  student.work() // undefined...work |

## 原型链+对象冒充的组合继承

|  |
| --- |
| 原型链+对象冒充的组合继承：实例化子类的时可以给父类传参，也可以继承原型上的属性和方法  *function* Person(*name*, *age*){    this.name = *name*    this.age = *age*    this.run = *function*() {        console.log(this.name + '...run')    }  }  Person.prototype.sex = '男'  Person.prototype.work = *function*() {    console.log(this.name + '...work')  }  *function* Student(*name*,*age*) {    //对象冒充继承  可以继承构造函数里面的属性和方法、实例化子类可以给父类传参    Person.call(this, *name*, *age*)  }  Student.prototype = new Person()  // 实例化子类可以给父类传参,  *let* student = new Student('张三',20)  student.run() // 张三...run  // 可以继承原型链上的属性和方法  student.work() // 张三...work |

## 原型链+对象冒充继承的另一种方式

|  |
| --- |
| *function* Person(*name*, *age*) {    this.name = *name*    this.age = *age*    this.run = *function* () {      console.log(this.name + '...run')    }  }  Person.prototype.sex = '男'  Person.prototype.work = *function* () {    console.log(this.name + '...work')  }  *function* Student(*name*, *age*) {    //对象冒充继承  可以继承构造函数里面的属性和方法、实例化子类可以给父类传参    Person.call(this, *name*, *age*)  }  // 子类的原型指向父类的原型  Student.prototype = Person.prototype  *let* student = new Student('张三', 20)  student.run()  student.work() |

# typeScript类、继承

## 继承extends 、super

|  |
| --- |
| *class* Person{    name: *string*    //构造函数：实例化类时触发  *constructor*(*name*:*string*) {      this.name = *name*    }    run():*void* {      console.log(this.name, '...父类')    }  }  *let* p = new Person('王五')  p.run() // 王五 ...父类  *class* Student extends Person{  *constructor*(*name*:*string*){      // 初始化父类的构造函数      super(*name*)    }    // 子类和父类有相同方法时，子类调用自身的    run():*void* {      console.log(this.name, '...run子类')    }    work():*void* {      console.log(this.name, '...work子类')    }  }  *let* student = new Student('张三')  student.run() // 张三 ...run子类  student.work() // 张三 ...work子类 |

## 类的修饰符(访问权限)

|  |
| --- |
| public :公有 在当前类里面、 子类 、类外面都可以访问  protected：保护类型 在当前类里面、子类里面可以访问 ，在类外部没法访问  private ：私有 在当前类里面可以访问，子类、类外部都没法访问  属性如果不加修饰符 默认就是 公有 （public） |

### public

|  |
| --- |
| *class* Person {    public name:*string*  *constructor*(*name*:*string*){        this.name = *name*    }    run():*void* {      // 在父类里面可以访问      console.log(this.name, '...父类')    }  }  *let* p = new Person('王五')  p.run()  // 在父类外可以访问  console.log(p.name, '...父类')  *class* Student extends Person{  *constructor*(*name*:*string*){        super(*name*)    }    run():*void* {      // 在子类里面可以访问      console.log(this.name, '...run子类')    }  }  *let* student = new Student('张三')  student.run()  // 在子类外可以访问  console.log(student.name, '...run子类') |

### protected

|  |
| --- |
| *class* Person{    protected name:*string*  *constructor*(*name*:*string*){      this.name = *name*    }    run():*void* {      // 在父类里面可以访问      console.log(this.name, '...父类')    }  }  *let* p = new Person('王五')  p.run()  // 在父类外不可以访问  console.log(p.name, '...父类')  *class* Student extends Person{  *constructor*(*name*:*string*) {      super(*name*)    }    run():*void* {      // 在子类里面可以访问      console.log(this.name, '...run子类')    }  }  *let* student = new Student('张三')  student.run()  // 在子类外不可以访问  console.log(student.name, '...run子类') |

### private

|  |
| --- |
| *class* Person{    private name:*string*  *constructor*(*name*:*string*){      this.name = *name*    }    run():*void* {      // 在父类里面可以访问      console.log(this.name, '...父类')    }  }  *let* p = new Person('王五')  p.run()  // 在父类外不可以访问  console.log(p.name, '...父类')  *class* Student extends Person{  *constructor*(*name*:*string*) {      super(*name*)    }    run():*void* {      // 在子类里面不可以访问      console.log(this.name, '...run子类')    }  }  *let* student = new Student('张三')  student.run()  // 在子类外不可以访问  console.log(student.name, '...run子类') |

## 静态属性 静态方法

### ES5 静态属性 静态方法

|  |
| --- |
| *function* Person(*name*) {    // 实例属性    this.name1 = *name*    // 实例方法    this.run = *function*() {      console.log(this.name1 + '...实例方法')    }  }  // 静态属性  Person.name2 = '李四'  // 静态方法  Person.run = *function*() {    console.log(Person.name2 + '...静态方法')  }  *let* p = new Person('张三')  // 调用实例方法、实例属性  p.run() // 张三...实例方法  console.log(p.name1) // 张三  // 调用静态方法、静态属性  Person.run() // 李四...静态方法  console.log(Person.name2) // 李四 |

### typeScript静态属性 静态方法

|  |
| --- |
| *class* Person{    // 属性    public name:*string*    // 静态属性    static sex="男"  *constructor*(*name*:*string*) {      this.name = *name*    }    run():*void* {      console.log(this.name + '...run') // 张三...run      // 实例方法可以使用静态属性      console.log(Person.sex + '...run') // 男...run    }      static print() {      // 静态方法，不能直接调用类里面的属性      // console.log(this.name + '...print')      console.log(Person.sex+ '...print') // 男...print    }  }  *let* p:Person = new Person('张三')  p.run()  // 类名直接调用静态方法、静态属性  Person.print()  console.log(Person.sex) // 男 |

## 多态

|  |
| --- |
| // 多态: 父类定义一个方法不去实现，让继承它的子类去实现  每一个子类有不同的表现  // 多态属于继承  *class* Animal {    name:*string*  *constructor*(*name*:*string*) {        this.name= *name*    }    // 继承它的子类去实现 ，每个子类吃什么，不一样    eat() :*void*{        console.log('吃的方法')    }  }  *class* Dog extends Animal {  *constructor*(*name*:*string*){        super(*name*)    }    eat():*void* {       console.log(this.name + '吃狗粮')    }  }  *class* Cat extends Animal{  *constructor*(*name*:*string*) {        super(*name*)    }    eat():*void*{        console.log(this.name + '吃猫粮')    }  }  *let* dog = new Dog('狗')  dog.eat()  *let* cat = new Cat('猫')  cat.eat() |

## 抽象类

|  |
| --- |
| /\*  abstract关键字定义抽象类和抽象方法  1.抽象类是提供其他类继承的基类，不能直接被实例化  2.抽象方法只能放在抽象类里面  3.抽象方法不包含具体实现，且必须在派生类中实现  4.非抽象方法，派生类可以不实现  5.抽象类和抽象方法用来定义标准  \*/  //标准：Animal 这个类要求它的子类必须包含eat方法  abstract *class* Animal{    public name:*string*  *constructor*(*name*:*string*) {        this.name = *name*    }    // 抽象方法    abstract eat():*any*    // 非抽象方法    run(): *void*{      console.log('非抽象方法可以不实现')    }  }  // let animal = new Animal() // 错误  *class* Dog extends Animal {    *constructor*(*name*:*any*){        super(*name*)    }    // 抽象类的子类必须实现抽象类里的抽象方法    eat():*void* {      console.log(this.name + '吃狗粮')    }  }  *class* Cat extends Animal{  *constructor*(*name*:*any*){        super(*name*)    }    eat():*void* {      console.log(this.name + '吃猫粮')   }  }  *let* dog = new Dog('狗')  dog.eat()  *let* cat = new Cat('猫')  cat.eat() |

# 接口

|  |
| --- |
| 接口的作用：在面向对象的编程中，接口是一种规范的定义，它定义了行为和动作的规范，  在程序设计里面，接口起到一种限制和规范的作用。接口定义了某一批类所需要遵守的规范，  接口不关心这些类的内部状态数据，也不关心这些类里方法的实现细节，  它只规定这批类里必须提供某些方法，提供这些方法的类就可以满足实际需要。  typescrip中的接口类似于java，同时还增加了更灵活的接口类型，包括属性、函数、可索引和类等。  接口：行为和动作的规范  抽象类也是定义行为和动作的规范，但只适应于类，  接口对属性、函数、类等都可以定义行为和动作的规范 |

## 属性接口 (对json的约束)

|  |
| --- |
| // 自定义方法的参数, 对json进行约束  *function* printLabel(*labelInfo*:{label:*string*}):*void* {    console.log('printLabel')  }  printLabel('hahah')      // 错误写法  printLabel({name:'张三'})  // 错误的写法  printLabel({label:'张三'})  // 正确的写法  ====================================  // 对批量方法的参数(对象)进行约束  *interface* FullName{    //注意;结束    firstName:*string*;    secondName:*string*;  }  // 传入的参数必须包含 firstName  secondName  *function* printName(*name*:FullName) {    console.log(*name*.firstName+'--'+*name*.secondName)  }  // printName('10')  // 错误  --------------------------------------------------  // 定义变量obj传入，不报错  *let* obj = {    age:20,    firstName:'张',    secondName:'三'  }  printName(obj)  --------------------------------------------------  // 参数的顺序可以不一样  printName({      secondName:'secondName',      firstName:'firstName'  })  --------------------------------------------------  //  直接传入对象报错，只能传入firstName  secondName  printName({    age:20,    firstName:'张',    secondName:'三'  })    --------------------------------------------------  //  传入的参数必须包含 firstName  secondName  *function* printInfo(*info*:FullName) {    console.log(*info*.firstName+'--'+*info*.secondName)  }  printInfo({    firstName:'李',    secondName:'四'  }) |

## 可选属性

|  |
| --- |
| *interface* FullName{      firstName:*string*;      secondName?:*string*;  }  *function* getName(*name*:FullName){      console.log(*name*)  }  getName({      firstName:'firstName'  }) |

## 函数类型接口

|  |
| --- |
| 函数类型接口:对方法传入的参数，以及返回值进行约束 批量约束  // 加密的函数类型接口  //1.对多个函数进行约束  *interface* encrypt {      (*key*:*string*,*value*:*string*):*string*;  }  *let* md5:encrypt = *function*(*key*:*string*,*value*:*string*):*string* {      //模拟操作      return *key* + *value*  }  console.log(md5('name','zhangsan'))  *let* sha1:encrypt=*function*(*key*:*string*,*value*:*string*):*string* {      //模拟操作      return *key* + *value*  }  console.log(sha1('name','lisi'))    ===============================================  // 2.对单个函数约束  *let* fun:(*n*: *string*, *a*:*number*) *=>* *string* = *function*(*n*: *string*, *a*:*number*):*string* {     return 'OK'  } |

## 可索引接口(不常用)

### 对数组的约束

|  |
| --- |
| *interface* UserArr {      [*index*:*number*]:*string*  }  *let* arr1:UserArr=[123, 'bbb']  // 错误  *let* arr2:UserArr=['aaa','bbb']  console.log(arr2[0]) |

### 对对象的约束

|  |
| --- |
| *interface* UserObj{      [*index*:*string*]:*string*  }  *let* obj:UserObj = {name:'张三'} |

## 类类型接口

|  |
| --- |
| //类类型接口: 对类的约束  和抽象类抽象有点相似  *interface* Animal {      name:*string*      eat(*str*:*string*):*void*  }  *class* Dog implements Animal{      name:*string*  *constructor*(*name*:*string*) {          this.name=*name*      }      eat(){          console.log(this.name + '吃粮食')      }  }  *let* dog = new Dog('小黑')  dog.eat() |

## 接口继承接口

|  |
| --- |
| *interface* Animal{    eat():*void*  }  *interface* Person extends Animal{    work():*void*  }  *class* Programmer{    public name:*string*  *constructor*(*name*:*string*){        this.name = *name*    }      coding(*code*:*string*) {        console.log(this.name + *code*)    }  }  *class* Student extends Programmer implements Person{    *constructor*(*name*:*string*){       super(*name*)    }    eat():*void* {      console.log(this.name+'吃馒头')    }    work():*void* {       console.log(this.name+'工作')    }  }  *let* student = new Student('小李')  student.eat()  student.work()  student.coding('写代码')    =================  *interface* Shape{    color:*string*;  }  *interface* PenStroke{   penWidth:*number*  }  *interface* Square extends Shape, PenStroke {    sidleLength:*number*    // reset():void;    // (start:number):string;    }  *let* s = <Square>{}  s.color = 'blue'  s.sidleLength = 10  s.penWidth = 10  console.log(s) |

# 泛型

|  |
| --- |
| 泛型就是解决 类 接口 方法的复用性、以及对不特定数据类型的支持(类型校验) |

## 泛型函数

|  |
| --- |
| // 同时返回 string和number类型，any可以解决这个问题  // any放弃了类型检查,传入什么 返回什么  *function* getData(*value*:*any*):*any* {    return *value*  }  getData(10)  getData('hello')  ==============================================  // any问题：参数类型和返回的参数类型可以不一致  *function* getData(*value*:*any*):*any* {      return '哈哈哈'  }  ===============================================  // T表示泛型，具体什么类型是调用这个方法的时候决定的  *function* getData<T>(*value*:T):T {      return *value*  }  getData<*number*>(10);  getData<*number*>('10')    // 错误  getData<*string*>('10')  ========================================  *function* Hello1<T>(*num*:T):T {      //  console.log(num.length) //报错       return *num*   }    *function* Hello2<T>(*str*:T[]):T[] {       console.log(*str*.length) //通过       return *str*   }  *let* list:*Array*<*string*> = Hello2<*string*>(['1','2','3'])   --------------------------------------  *function* Hello<T>(*arg*:T):T{      return *arg*   }  *let* myHello1:<K>(*arg*:K) *=>* K = Hello   console.log(myHello1('hello'))     // 或  *let* myHello2:{<T>(*arg*:T)} = Hello   console.log(myHello2('hello'))  ==========================================  *let* myFunc:(*a*:*number*) *=>* *string* = *function*(*a*:*number*):*string* {     return 'hello' + *a*  }  console.log(myFunc(2))  ===========================================  *interface* Hello3{      <T>(*arg*:T):T   }  *function* myHello3<T>(*arg*:T):T {      return *arg*   }  *let* MH:Hello3 = myHello3   console.log(MH('hello'))   console.log(MH<*string*>('hello')) |

## 泛型类

|  |
| --- |
| // 泛型类：比如有个最小堆算法，需要同时支持返回数字和字符串a-z两种类型。  *class* MinClas<T>{      public list:T[] = []      add(*value*:T):*void* {          this.list.push(*value*)      }      min():T {  *let* minNum = this.list[0]          for(*let* i=0; i<this.list.length; i++) {              if(minNum>this.list[i]) {                  minNum=this.list[i]              }          }          return minNum      }  }  // 实例化类 并且制定了类的T代表的类型是number  *let* m1=new MinClas<*number*>()  m1.add(11);  m1.add(3);  m1.add(2);  console.log(m1.min())  // 实例化类 并且制定了类的T代表的类型是string  *let* m2=new MinClas<*string*>()  m2.add('c')  m2.add('a')  m2.add('v')  console.log(m2.min())  ======================================  *class* HelloNumber<T> {      ten:T      add:(*x*:T,*y*:T) *=>* T   }  *let* myHelloNumber = new HelloNumber<*number*>()   myHelloNumber.ten = 10   myHelloNumber.add = *function*(*x*,*y*) {      return *x*+*y*   }   console.log(myHelloNumber.ten)   console.log(myHelloNumber.add(myHelloNumber.ten, 20))  ==============================  *class* Person<T> {     }  *class* UserInfo {      name:*string* = '张三'      age:*number* = 16   }  *let* a = new Person<UserInfo>()    =============================================  *interface* School<T> {      add(*n*:T):*boolean*   }  *class* Student1 implements School<UserInfo> {      add(*n*:UserInfo):*boolean* {          return true      }   } |

## 泛型接口

|  |
| --- |
| // 函数类型接口  *interface* ConfigFn {      (*value1*:*string*,*value2*:*string*):*string*  }  *let* setData:ConfigFn = *function*(*value1*:*string*,*value2*:*string*):*string* {      return *value1*+*value2*  }  setData('name','张三')  ========================================  //1、泛型接口  *interface* ConfigFn {      <T>(*value*:T):T  }  *let* getData:ConfigFn = *function*<T>(*value*:T):T {      return *value*  }  getData<*string*>('张三')  getData<*string*>(10)  //错误  ========================================  //2、泛型接口  *interface* ConfigFn<T> {      (*value*:T):T  }  *function* getData<T>(*value*:T):T {      return *value*  }  *let* myGetData:ConfigFn<*string*> = getData  myGetData('20')  myGetData(20)  //错误 |

## 泛型实现mySql的访问

|  |
| --- |
| *interface* IData<T> {      Add(*info*:T):*boolean*      Delete(*info*:T):*boolean*      Update(*info*:T):T      Search(*info*:T, *id*:*number*):T   }     // 对mySql的访问, 泛型实现泛型  *class* MysqlData<T> implements IData<T> {      Add(*info*: T): *boolean* {          return false      }      Delete(*info*: T): *boolean* {          return false      }      Update(*info*: T): T {          return *info*      }      Search(*info*:T, *id*: *number*): T {          return *info*      }   }   // 对msSql的访问,  *class* MssqlData<T> implements IData<T> {      Add(*info*: T): *boolean* {          return false      }      Delete(*info*: T): *boolean* {          return false      }      Update(*info*: T): T {          return *info*      }      Search(*info*:T, *id*: *number*): T {          return *info*      }   }    *class* UserInfo1 {     }     // 调用，传入具体类型  *class* UserData extends MysqlData<UserInfo1> {      public GetUserInfo(*n*:*number*):UserInfo1 {  *let* info = new UserInfo1()          return this.Search(info, *n*)      }   }    *let* ud = new UserData()   console.log(ud.GetUserInfo(1)) |

## 类作为参数来约束数据传入的类型

### 常规做法

|  |
| --- |
| 1.定义一个User的类，这个类的作用就是映射数据库字段  2.定义一个 MysqlDb的类，这个类用于操作数据库  3.然后把User类作为参数传入到MysqlDb中  问题：下面MysqlDb类代码重复了  *class* User {    username:*string* | *undefined*    pasword:*string* | *undefined*  }  *class* MysqlDb {    add(*user*:User):*boolean*{        console.log(*user*)        return true    }  }  *let* u = new User()  u.username = '张三'  u.pasword = '123456'  *let* db = new MysqlDb()  db.add(u)  =============================================================  *class* ArticleCate {    title:*string* | *undefined*    desc:*string* | *undefined*    status:*number* | *undefined*  }  *class* MysqlDb {    add(*info*:ArticleCate):*boolean*{        console.log(*info*)        return true    }  }  *let* a = new ArticleCate()  a.title = '国内'  a.desc = '国内新闻'  a.status = 1  *let* db = new MysqlDb()  db.add(a) |

### 泛型类做法

|  |
| --- |
| // 定义操作数据库的泛型类  *class* MysqlDb<T> {    add(*info*:T):*boolean* {      console.log(*info*)      return true    }    updated(*info*:T, *id*:*number*):*boolean* {      console.log(*info*, *id*)      return true    }  }  // 1、定义一个User类 和数据库进行映射  *class* User {    username:*string* | *undefined*    pasword:*string* | *undefined*  }  *let* u = new User()  u.username = '张三'  u.pasword = '123456'  *let* db1 = new MysqlDb<User>()  // 给User表增加数据  db1.add(u)  ===================================================  //2、相关ArticleCate增加数据  定义一个ArticleCate类 和数据库进行映射  *class* ArticleCate {    title:*string* | *undefined*    desc:*string* | *undefined*    status:*number* | *undefined*  *constructor*(*params*: {      title:*string* | *undefined*,      desc:*string* | *undefined*,      status?:*number* | *undefined*    }) {        this.title=*params*.title        this.desc=*params*.desc        this.status=*params*.status    }  }  //增加操作  *let* a1 = new ArticleCate({    title: '分类1',    desc: '1111',    status: 1  })  // 类当做参数的泛型类  *let* db2 = new MysqlDb<ArticleCate>()  db2.add(a1)  //修改数据  *let* a2=new ArticleCate({    title:'分类2',    desc:'2222'  })  a2.status = 0  *let* db3 = new MysqlDb<ArticleCate>()  db3.updated(a2, 12) |

## 封装统一操作Mysql Mongodb Mssql的底层类库

|  |
| --- |
| /\*  功能：定义一个操作数据库的库  支持 Mysql Mssql  MongoDb  要求1：Mysql MsSql  MongoDb功能一样  都有 add  update  delete  get方法  注意：约束统一的规范、以及代码重用  解决方案：需要约束规范所以要定义接口 ，需要代码重用所以用到泛型      1、接口：在面向对象的编程中，接口是一种规范的定义，它定义了行为和动作的规范      2、泛型 通俗理解：泛型就是解决 类 接口 方法的复用性、  \*/  *interface* DBI<T> {      add(*info*:T):*boolean*;      update(*info*:T,*id*:*number*):*boolean*;      delete(*id*:*number*):*boolean*;      get(*id*:*number*):*any*[];  }  // 定义一个操作mysql数据库的类  注意：要实现泛型接口 这个类也应该是一个泛型类  *class* MysqlDb<T> implements DBI<T>{  *constructor*(){          console.log('数据库建立连接')      }      add(*info*: T): *boolean* {          console.log(*info*);          return true;      }      update(*info*: T, *id*: *number*): *boolean* {          throw new *Error*("Method not implemented.");      }      delete(*id*: *number*): *boolean* {          throw new *Error*("Method not implemented.");      }      get(*id*: *number*): *any*[] {  *let* list=[              {                  title:'标题1',                  desc:'哈哈哈哈哈'              },              {                  title:'标题2',                  desc:'嘻嘻嘻嘻嘻'              }          ]          return list;      }  }  //定义一个操作mssql数据库的类  *class* MsSqlDb<T> implements DBI<T>{  *constructor*(){          console.log('数据库建立连接');      }      add(*info*: T): *boolean* {          console.log(info);          return true;      }      update(*info*: T, *id*: *number*): *boolean* {          throw new *Error*("Method not implemented.");      }      delete(*id*: *number*): *boolean* {          throw new *Error*("Method not implemented.");      }      get(*id*: *number*): *any*[] {  *let* list=[              {                  title:'标题1',                  desc:'哈哈哈哈哈'              },              {                  title:'标题2',                  desc:'嘻嘻嘻嘻嘻'              }          ]          return list;      }  }  //操作用户表   定义一个User类和数据表做映射  *class* User1{      username:*string* | *undefined*;      password:*string* | *undefined*;  }  *let* u1 = new User1()  u1.username='张三111'  u1.password='123456'  // 类作为参数来约束数据传入的类型  *let* oMysql=new MysqlDb<User1>()  // 增加  oMysql.add(u1)  *class* User2 {      username:*string* | *undefined*;      password:*string* | *undefined*;  }  *let* u2=new User2();  u2.username='张三2222';  u2.password='123456';  *let* oMssql=new MsSqlDb<User2>();  // 增加  oMssql.add(u2);  // 获取User表 ID=4的数据  *let* data = oMssql.get(4);  console.log(data); |

# 模块

|  |
| --- |
| //  javaScrip Module模块  //  1.模块化、可复用  //  2.封装变量和函数   (*function*() {      // 内部代码   })()  *let* a// 全局  *function* hello() {  *let* b// 局部      c //全局   }     (*function*(*$*, *w*){     })(jQuery, window)    *module* Validation {      export *interface* StringValidator {          isAcceptable(*s*:*string*): *boolean*      }    *let* letterRegexp = /^[A-Za-z]+$/  *let* numberRegexp = /^[0-9]+$/  *class* LettersOnValidator implements StringValidator {          isAcceptable(*s*:*string*):*boolean* {              return letterRegexp.test(*s*)          }      }    *class* ZipOnValidator implements StringValidator {          isAcceptable(*s*:*string*):*boolean* {              return numberRegexp.test(*s*)          }      }   }     // 模块应用  *module* Time {      export *class* Test {          element: *HTMLElement*          span:*HTMLElement*          timer:*number*  *constructor*(*e*:*HTMLElement*) {              this.element = *e*              this.element.innerHTML = '时间：'              this.span = document.createElement('span')              this.element.appendChild(this.span)              this.span.innerHTML = new *Date*().toTimeString()          }          start() {             this.timer = setInterval(() *=>* {                 return this.span.innerHTML = new *Date*().toTimeString()             })          }          stop() {              clearInterval(this.timer)          }      }   }     // index.ts  *let* div = document.createElement('div')   document.body.appendChild(div)  *let* obj = new Time.Test(div)    *let* button = document.createElement('button')   button.innerHTML = 'start'   button.onclick = *function*() {      obj.start()   }   document.body.appendChild(button)    *let* buttons = document.createElement('button')   buttons.innerHTML = 'stop'   buttons.onclick = *function*() {      obj.stop()   }   document.body.appendChild(buttons) |

# 命名空间

|  |
| --- |
| /\*  命名空间:      在代码量较大的情况下，为了避免各种变量命名相冲突，可将相似功能的函数、类、接口等放置到命名空间内      同Java的包、.Net的命名空间一样，TypeScript的命名空间可以将代码包裹起来，只对外暴露需要在外部访问的对象。      命名空间内的对象通过export关键字对外暴露。  命名空间和模块的区别：      命名空间：内部模块，主要用于组织代码，避免命名冲突。      模块：ts的外部模块的简称，侧重代码的复用，一个模块里可能会有多个命名空间。  \*/  animal.ts  export *namespace* A{  *interface* Animal {          name: *string*;          eat(): *void*;      }      export *class* Dog implements Animal {          name: *string*;  *constructor*(*theName*: *string*) {              this.name = *theName*;          }          eat() {              console.log(`${this.name} 在吃狗粮`);          }      }      export *class* Cat implements Animal {          name: *string*;  *constructor*(*theName*: *string*) {              this.name = *theName*;          }          eat() {              console.log(`${this.name} 吃猫粮`);          }      }  }  export *namespace* B {  *interface* Animal {          name: *string*;          eat(): *void*;      }      export *class* Dog implements Animal {          name: *string*;  *constructor*(*theName*: *string*) {              this.name = *theName*;          }          eat() {              console.log(`${this.name} 在吃狗粮`);          }      }      export *class* Cat implements Animal {          name: *string*;  *constructor*(*theName*: *string*) {              this.name = *theName*;          }          eat() {              console.log(`${this.name} 在吃猫粮`);          }      }  }  index.ts  import {A,B} from './modules/animal'  *let* dog1 = new A.Dog('小黑')  dog1.eat()  *let* dog2 = new B.Dog('小花')  dog2.eat()  =======================================  *namespace* Validation {      export *interface* StringValidator {          isAcceptable(*s*: *string*): *boolean*      }    *const* lettersRegexp = /^[A-Za-z]+$/  *const* numberRegexp = /^[0-9]+$/        export *class* LettersOnlyValidator implements StringValidator {          isAcceptable(*s*: *string*) {              return lettersRegexp.test(*s*)          }      }        export *class* ZipCodeValidator implements StringValidator {          isAcceptable(*s*: *string*) {              return *s*.length === 5 && numberRegexp.test(*s*)          }      }   }     // demo.ts   /// <reference path="./Validate" />  *let* m1 = new Validate.StringValidate() |

# 装饰器

|  |
| --- |
| 装饰器:装饰器是一种特殊类型的声明，它能够被附加到类声明，方法，属性或参数上，可以修改类的行为。  通俗的讲装饰器就是一个方法，可以注入到类、方法、属性参数上来扩展类、属性、方法、参数的功能。  常见的装饰器有：类装饰器、属性装饰器、方法装饰器、参数装饰器  装饰器的写法：普通装饰器（无法传参） 、 装饰器工厂（可传参）  装饰器是过去几年中js最大的成就之一，已是Es7的标准特性之一 |

## 类装饰器

### 扩展当前类功能

|  |
| --- |
| // 1.类装饰器:普通装饰器（无法传参）  *function* logClass(*params*:*any*) {    // params 就是当前HttpClient类    console.log(*params*)    // 给类添加属性和实例方法  *params*.prototype.apiUrl = '动态扩展的属性'  *params*.prototype.run = *function*() {        console.log('run方法')    }  }  @logClass  *class* HttpClient {  *constructor*() {    }    getData() {    }  }  *let* http:*any* = new HttpClient()  console.log(http.apiUrl)  http.run()    ===========================================  // 2 类装饰器:装饰器工厂（可传参）  *function* logClass(*params*: *string*) {    return *function*(*target*:*any*) {      // target表示传入的HttpClient类      console.log(*target*)       // params表示传入的参数      console.log(*params*)  *target*.prototype.apiUrl = *params*    }  }  @logClass('http://www.itying.com/api')  *class* HttpClient {  *constructor*() {    }    getData() {    }  }  *let* http:*any* = new HttpClient()  console.log(http.apiUrl) |

### 修改当前类属性、构造函数、方法

|  |
| --- |
| /\*  1、类装饰器    下面是一个重载构造函数的例子。    类装饰器会在运行时当作函数被调用，类的构造函数作为其唯一的参数。    如果类装饰器返回一个值， 该值(构造函数)会替换 HttpClient类的声明  \*/  // 在装饰器里重载(修改)HttpClient的属性 构造函数、方法  *function* logClass(*target*:*any*) {    console.log(*target*)    return *class* extends *target* {      // 修改HttpClient类里的apiUrl      apiUrl:*string* = '装饰器的apiUrl'  *constructor*() {        super()        this.apiUrl = '装饰器的apiUrlxx'        console.log('装饰器：构造函数执行')      }      // 修改HttpClient类里的getData      getData() {        console.log('装饰器：', this.apiUrl)        console.log('装饰器：getData方法执行')      }    }  }  @logClass  *class* HttpClient {      public apiUrl:*string* | *undefined*  *constructor*() {        this.apiUrl = 'HttpClient的apiUrl'        console.log('HttpClient：构造函数执行')      }      getData() {        console.log('HttpClient：', this.apiUrl)        console.log('HttpClient：getData方法执行')      }  }  *let* http = new HttpClient()  // 调用装饰器里的getData  http.getData()  // 装饰器里的apiUrl  console.log(http.apiUrl) |

## 属性装饰器

|  |
| --- |
| 属性装饰器会在运行时当作函数被调用，传入下列2个参数：    1、对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。    2、成员的名字。  // 类装饰器  *function* logClass(*params*:*string*){    return *function*(*target*:*any*){        console.log('类装饰器：', *target*)        console.log('类装饰器：', *params*)    }  }  // 属性装饰器  *function* logProperty(*params*:*any*) {    return *function*(*target*:*any*, *attr*:*any*) {  *target*[*attr*] = *params*      // params传入的参数  http://itying.com      console.log('属性装饰器：', *params*)      // target对于实例成员是类的原型对象      console.log('属性装饰器：', *target*)      // attr成员的名字 url      console.log('属性装饰器：', *attr*)      // http://itying.com      console.log('属性装饰器：', *target*[*attr*])    }  }  @logClass('https://www.baidu.com')  *class* HttpClient {    @logProperty('http://itying.com')    public url:*any* | *undefined*  *constructor*() {      console.log('HttpClient构造函数')    }    getData() {      console.log('HttpClient：', this.url)    }  }  *let* http = new HttpClient()  http.getData() |

## 方法装饰器

|  |
| --- |
| 方法装饰器被应用到方法的 属性描述符上，可以用来监视，修改、替换方法定义。  方法装饰器会在运行时传入下列3个参数：      1、对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。      2、成员的名字。      3、成员的属性描述符。 |

### 方法装饰器一

|  |
| --- |
| *function* get(*params*:*any*) {    return *function*(*target*:*any*,*methodName*:*any*,*desc*:*any*) {      // 参数      console.log(*params*)      // 对于实例成员是类的原型对象      console.log(*target*)      // 成员的名字      console.log(*methodName*)      // 成员的属性描述符      console.log(*desc*)  *target*.apiUrl='https://www.baidu.com'  *target*.run = *function*() {          console.log('run方法')      }    }  }  *class* HttpClient {    public url:*any* | *undefined*  *constructor*(){    }    @get('http://www.itying,com')    getData(){        console.log(this.url)    }  }  *let* http:*any* = new HttpClient()  console.log(http.apiUrl) //https://www.baidu.com  http.run() |

### 方法装饰器二

|  |
| --- |
| *function* get(*params*:*any*) {    return *function*(*target*:*any*, *methodName*:*any*, *desc*:*any*) {      // 参数      console.log(*params*)      // 对于实例成员是类的原型对象      console.log(*target*)      // 成员的名字      console.log(*methodName*)      // 成员的属性描述符      console.log(*desc*)      console.log(*desc*.value)        // 保存当前的方法  *let* oMethod = *desc*.value      // 修改装饰器的方法(desc.value)，把传入装饰器里参数改为string类型  *desc*.value = *function*(...*args*:*any*[]) {  *args* = *args*.map((*value*) *=>* {          return *String*(*value*)        })        oMethod.apply(this,*args*)      }    }  }  *class* HttpClient {      public url:*any* | *undefined*  *constructor*(){      }      @get('http://www.itying,com')      getData(...*args*:*any*[]) {        console.log('HttpClient：', *args*)        console.log('HttpClient里面的getData方法')      }  }  *let* http = new HttpClient()  http.getData(123, 'https://www.baidu.com') |

## 参数装饰器

|  |
| --- |
| 参数装饰器表达式会在运行时当作函数被调用，可以使用参数装饰器为类的原型增加一些元素数据 ，传入下列3个参数：  1、对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。  2、方法的名字。  3、参数在函数参数列表中的索引。  *function* logParams(*params*:*any*){    return *function*(*target*:*any*,*methodName*:*any*,*paramsIndex*:*any*){      // 参数      console.log(*params*)      // 对于实例成员是类的原型对象      console.log(*target*)      // 方法的名字      console.log(*methodName*)      // 参数在函数参数列表中的索引      console.log(*paramsIndex*)      // 修改HttpClient属性  *target*.apiUrl = *params*    }  }  *class* HttpClient {    public url:*any* |*undefined*  *constructor*() {      console.log('HttpClient构造函数')    }    getData(@logParams('https://www.baidu.com') *uuid*:*any*){      console.log(*uuid*)      console.log('HttpClient的getData方法')    }   }  *let* http:*any* = new HttpClient()  http.getData(123456)  //  https://www.baidu.com  console.log( http.apiUrl) |

## 装饰器执行顺序

|  |
| --- |
| 1.属性》方法》方法参数》类  2.如果有多个同样的装饰器，它会先执行后面的  *function* logClass1(*params*:*string*) {    return *function*(*target*:*any*) {      console.log('类装饰器1')    }  }  *function* logClass2(*params*:*string*) {    return *function*(*target*:*any*) {      console.log('类装饰器2')    }  }  *function* logAttribute1(*params*?:*string*) {    return *function*(*target*:*any*,*attrName*:*any*) {      console.log('属性装饰器1')    }  }  *function* logAttribute2(*params*?:*string*) {    return *function*(*target*:*any*,*attrName*:*any*) {      console.log('属性装饰器2')    }  }  *function* logMethod1(*params*?:*string*) {    return *function*(*target*:*any*,*attrName*:*any*,*desc*:*any*) {      console.log('方法装饰器1')    }  }  *function* logMethod2(*params*?:*string* ){    return *function*(*target*:*any*,*attrName*:*any*,*desc*:*any*) {      console.log('方法装饰器2')    }  }  *function* logParams1(*params*?:*string*){    return *function*(*target*:*any*,*attrName*:*any*,*desc*:*any*){      console.log('方法参数装饰器1')    }  }  *function* logParams2(*params*?:*string*){    return *function*(*target*:*any*,*attrName*:*any*,*desc*:*any*){      console.log('方法参数装饰器2')    }  }    @logClass1('http://www.itying.com/api')  @logClass2('https://www.baidu.com')  *class* HttpClient {    @logAttribute1()    @logAttribute2()    public apiUrl:*string* | *undefined*  *constructor*() {      console.log('构造函数')    }    @logMethod1()    @logMethod2()    getData(){      return true    }    setData(@logParams1() *attr1*:*any*, @logParams2() *attr2*:*any*): *void* {      console.log('setData方法')    }  }  *let* http:*any* = new HttpClient()  http.setData() |

|  |
| --- |
|  |

|  |
| --- |
|  |